Introduction à Scala 3
Projet programmation 2

Vincent Lafeychine Yohan Géran Stefan Schwoon

25 janvier 2024

Ce document vous propose une introduction au langage de programmation Scala.

Certaines parties de ce document font référence à des liens de la documentation officielle de Scala :

- Les chapitres sont liés à des chapitres du *Book*, se présentant sous la forme de leçons.
- Les sous-chaptires sont liés à des pages du *Tour*, se présentant sous la forme de fiches concises.
- Les fonctions présentées sont liées à des pages de l'`API`.

Ce document ne propose qu'un aperçu des éléments qui nous paraissent essentiels à découvrir. Ainsi, n'hésitez pas à explorer davantage la documentation officielle et à nous poser des questions si besoin.

Table des matières

Pour votre projet, vous allez avoir besoin d'un compte GitLab auprès du CRANS. Pour cela, avant de poursuivre la lecture de ce document, veuillez vous inscrire sur le site intranet du CRANS en renseignant votre nom, prénom, adresse mail et pseudo. Ensuite, demander l'accès au GitLab, en précisant votre pseudo, à l'adresse mail `pigeonmoelleux@crans.org`. C'est un 3A Info, ne vous prenez pas trop la tête pour rédiger ce mail.

1 Le langage Scala 3 et `sbt`

(Book : A Taste of Scala — A First Look at Types — Control structures)

Scala est un langage de programmation objet construit par dessus Java, tout ce qui est disponible en Java l'est aussi en Scala. Contrairement à Java il est également possible d'écrire dans un style fonctionnel, ce qui permet d'améliorer grandement la lisibilité du code.

1.1 `sbt` (Simple Build Tool)

(Site internet de `sbt`)

`sbt` est un outil pour compiler et lancer efficacement du code Scala (ou Java).

Pour l'utiliser, veuillez respecter la hiérarchie de dossiers suivante :

```
src/
    main/
        scala/
            yourfiles.scala
        resources/
            (yourimages.png)
```

Le fichier `build.sbt` contient les options de compilation, il contiendra la version du langage Scala :

```
scalaVersion := "3.3.1"
```

`sbt` ne recompile que les sources qui ont été modifiées, ce qui fait gagner beaucoup de temps.

Vous mettrez vos ressources auxiliaires (comme les *sprites*) dans le dossier `resources/`.

Écrivez un fichier `Main.scala` dans le dossier `src/main/scala/` contenant :

```
@main def main =
    println("Hello world!")
```

Vous lancerez la commande `sbt` à la racine de votre projet, ce qui lancera un interpréteur.

Cet interpréteur vous permet de :

- compiler votre projet avec `compile` ;
- lancer votre projet avec `run` ;
- ouvrir un *top-level* interactif (`REPL`) avec `console`.

En entrant `run` dans l'interpréteur, `sbt` va automatiquement :

- installer la bonne version du compilateur Scala et toutes les dépendances nécessaires ;
- vérifier si des changements ont eu lieu dans votre projet et compiler si besoin, puis ;
- lancer votre projet et afficher `"Hello world!"`.

Il est conseillé de laisser l'interpréteur ouvert lorsque vous développerez votre projet, `sbt` étant assez lent à s'initialiser.

1.2 Introduction et syntaxe (Tour : Basics)

Pour cette section, vous pouvez utiliser le `REPL` avec la commande `console`.

Recopiez ligne par ligne les commandes ci-dessous en observant leur résultat :

```
1 + 1
```

Pour déclarer une nouvelle variable mutable, on utilise le mot clé `var` :

```
var x = 0
x = x + 1
x += 1
```

On peut déclarer des valeurs immuables avec `val` :

```
val y = 0
y = y + 1
```

On définit une fonction avec `def` :

```
def plusOne(n: Int): Int = n + 1

plusOne(1)
```

On peut également omettre le type de sortie, et Scala l'inférera :

```
def plusOne(n: Int) = n + 1

plusOne(1)
```

Cependant, nous vons encourageons à expliciter les types des fonctions pour plus de lisibilité.

Une exception est faite lorsque votre fonction ne renvoie rien (`Unit`).

Pour des fonctions de plusieurs lignes, on utilisera la syntaxe suivante (similaire à Python) :

```
var x = 0

def count() =
    x += 1
    println("Counter: " + x)          // Concaténation avec l'opérateur +
    println(s"Counter: ${x}")         // Interpolation de chaînes de caractères
```

On remarque que lorsque la fonction ne prend pas d'argument, on peut omettre les parenthèses lors de son appel et de sa définition. Cela indique que la fonction n'effectue pas d'effets de bord, comme la mutation d'une donnée dans un objet (voir section 2 sur la programmation objet).

Une syntaxe des boucles `for` est la suivante :

```
for x <- 0 to 10 do
    println(x)
```

▶ *Écrivez une fonction calculant la factorielle d'un entier de deux manières différentes : avec une boucle et récursive dans le `REPL`, puis dans des fichiers Scala dans le dossier `src/main/scala`.*

2 Programmation orientée objet

(Book : Domain Modeling → OOP Modeling — Methods)

L'idée de base est d'identifier les « objets » manipulés par un programme et de structurer la programmation autour de ceux-ci. Un objet peut représenter un objet naturel qui interagit avec d'autres objets ou bien une structure de données avec ses opérations.

Quelques exemples pour des objets :

- dans une base de données : les tableaux, les requêtes, etc ;
- dans une interface graphique : les fenêtres, les boutons, etc ;
- dans un jeu vidéo : les différents acteurs.

Un objet possède des données mutables (`var`) et immutable (`val`) qui définissent l'état interne de l'objet et des *méthodes* (`def`). Les méthodes permettent d'interagir avec l'objet, elles peuvent modifier les données mutables, renvoyer de l'information et interagir avec d'autres objets.

L'exécution d'un programme consiste en l'interaction de ces objets, créés lors de cette exécution.

2.1 Classes

(Tour : Classes)

On regroupe des objets similaires dans une *classe*. Par exemple, dans une interface graphique, les composants d'une fenêtre, tel que des boutons cliquables, formeraient une classe. On écrit donc du code en décrivant le comportement des classes.

Comme exemple, nous allons prendre un vélo : on considère qu'il dispose d'un compteur kilométrique et qu'il permet de bouger et freiner.

```
class Bicycle:
    // Même si votre projet ne sera développé que par des francophones,
    // écrivez votre projet (noms de variables, documentation, ...) en utilisant de l'anglais.
    var counter: Double = 0

    def move() =
        counter += 1
        println(s"Counter: ${counter}")

    def brake() =
        println("I stop here!")
```

Dans la fonction `main`, on place le code suivant qui sert à pédaler 10 km.

```
@main def main =
    val b = Bicycle()

    while (b.counter < 10) do
        b.move()

    b.brake()
```

La première ligne de notre fonction `main` crée une instance de `Bicycle` en effectuant un appel avec le nom de la classe.

La fonction qui porte le même nom que la classe et qui permet de créer une nouvelle instance de cette classe s'appelle un constructeur de la classe.

Dans le paradigme objet, chaque instance possède ses propres données. Ainsi, nous pouvons créer deux vélos en même temps, chacun ayant son propre compteur :

```
@main def main =
    val b = Bicycle()
    val c = Bicycle()

    while (b.counter < 10) do
        b.move()

    while (c.counter < 5) do
        c.move()
```

Les objets peuvent prendre des paramètres lors de leur création :

```
class Bicycle(val name: String):
    ...

@main def main =
    val b = Bicycle("Moulinette")
```

Sur l'exemple ci-dessus, le constructeur de la classe `Bicycle` prend un paramètre `name`, qui sera nécessaire de fournir lors de la création de la classe.

Il est à noter que ces paramètres sont précédés de `var` (mutable) ou de `val` (immutable). Si un telle annotation n'est pas mise, le paramètre ne sera accessible que dans le corps de l'objet.

▶ *Faites afficher le nom du vélo dans `move`.*

2.2 Héritage (Tour : Classes)

Un aspect intéressant de la programmation orientée objet est le partage de code. Si certains objets d'une classe `C` ont des comportements différents ou supplémentaires par rapport aux autres membres, il convient de les regrouper dans une sous-classe en formant donc une hiérachie.

Dans une sous-classe, on ne décrit que les différences avec la super-classe. La terminologie « classe enfante » (sous-classe) et « classe parente » (super-classe) existe également.

Par exemple, considérons un vélo de route (qui est plus rapide) et un vélo rouillé (qui fait du bruit lors du freinage) en remplaçant les méthodes de la super-classe en utilisant le mot-clé `override` :

```
class RoadBicycle(name: String) extends Bicycle(name):
    override def move() =
        counter += 1.5
        println(s"${name}: *moving*")

class RustyBicycle(name: String) extends Bicycle(name):
    override def brake() =
        println("eek")
```

On peut toutefois réutiliser une fonction remplacée en utilisant le mot-clé `super`. Par exemple, il est possible d'écrire la méthode `move` dans `RoadBicycle` de la manière suivante :

```
override def move() =
    counter += 0.5
    super.move()
```

Une méthode qui accepte comme argument un objet d'une classe `C` peut travailler sur n'importe quelle sous-classe de `C`, tout en utilisant les méthodes remplacées de la sous-classe.

▶ *Regroupez les lignes concernant* ***`move`*** *et* ***`brake`*** *dans* ***`main`*** *dans une méthode* ***`travel`*** *qui prend comme paramètre un* ***`Bicycle`*** *et la distance à parcourir. Utilisez-la avec deux vélos différents.*

Toute classe possède automatiquement une méthode `toString`, la représentation canonique textuelle de chaque objet :

```
@main def main =
    val b = new Bicycle("Moulinette")
    println(b)
```

Par défaut, les objets affichent la classe dont ils sont issues, ainsi que leur adresse dans la mémoire.

▶ *Redéfinissez la méthode* ***`toString`*** *pour afficher le nom et le compteur d'une bicyclette.*

2.3 Modificateurs de visibilité (Tour : Classes)

Chaque classe doit être responsable des éléments et méthodes qui lui appartiennent.

Par exemple, le compteur `counter` de notre vélo ne devrait pas être accessible à la modification :

```
@main def main =
    val b = new Bicycle("Moulinette")
    b.counter = 42       // Il faut interdire cette modification.
    println(b.counter)   // Il faut autoriser cette lecture.
```

Il est possible d'écrire `counter` de façon immutable, mais cela nous empêcherait d'écrire `move` :

```
class Bicycle:
    val counter: Double = 0

    def move() =
        counter += 1     // Ne fonctionne plus...
```

Le paradigme objet offre la possibilité de restreindre la visibilité des données en utilisant les mots-clés :

- `private` : Empêchant à tout autre objet d'accéder à l'élément ;
- `protected` : Empêchant à tout autre objet d'accéder à l'élément, sauf aux sous-classes.

Il est à noter que ces mots-clés fonctionne sur les variables d'une classe mais également ses fonctions.

Une première étape est donc d'utiliser `protected` sur la variable `counter`, laissant la possibilité aux classes `RoadBicycle` et `RustyBicycle` de modifier la variable :

```
class Bicycle:
    protected val counter: Double = 0
```

Maintenant que la variable `counter` a une visibilité `protected`, il n'est plus possible d'accéder depuis l'extérieur de la classe :

```
@main def main =
    val b = new Bicycle("Moulinette")
    b.counter = 42       // Ne fonctionne pas: Parfait.
    println(b.counter)   // Ne fonctionne pas: Il faut autoriser cette lecture.
```

Pour remédier à cela, nous allons ajouter une fonction `counter`, appelée *getter*, à notre classe `Bicycle` qui permettra d'accéder à l'élément `counter` :

```
class Bicycle:
    // Tout accès de ‘counter’ passera maintenant par le nouveau getter.
    private val counter: Double = 0

    // On ne met pas de parenthèses pour cette fonction: Il n’y a pas d’effets de bord.
    def counter: Double = counter

    // Erreur: Double définition de ‘counter’ avec ‘var counter’ et ‘def counter’.
```

Le code ci-dessus ne va pas fonctionner, `counter` faisant référence à la fois à la variable et à notre nouvelle fonction *getter*. Il est commun de lever l'ambiguïté en préfixant la variable par un *underscore* :

```
class Bicycle:
    // Tout accès de ‘_counter’ passera maintenant par le nouveau getter.
    private val _counter: Double = 0

    // On ne met pas de parenthèses pour cette fonction: Il n’y a pas d’effets de bord.
    def counter: Double = _counter

    ...
```

Il est également possible de modifier la façon dont une variable est modifiée en définissant un *setter* :

```
class Bicycle:
    private var _counter: Double = 0

    def counter_=(newValue: Double): Unit =
        if newValue > 0 then
            _counter += newValue

    ...
```

Définir un *getter* et un *setter* est très courant en programmation objet afin de conserver les invariants que chaque classe souhaite préserver (comme l'exemple du *setter* de `counter` qui empêche `counter` d'être négatif). Faites donc bien attention aux attributs de visibilités de vos variables et méthodes !

2.4 Trait (Tour : Traits)

Un trait définit un ensemble de méthodes que des objets vont devoir implémenter, cela permet de regrouper des comportements communs.

Par exemple, un trait `Vehicle` générique regroupe des moyens de déplacements spécifiques comme les vélos ou les trains. Il ne sera pas possible d'instancier un véhicule, mais il sera possible d'instancier un vélo ou un train.

Un voyage se fait en bougeant jusqu'à ce qu'on ait parcouru une certaine distance. Considérons donc la déclaration suivante :

```
trait Vehicle:
    private val _counter: Double = 0

    def counter: Double = _counter

    def counter_=(newValue: Double): Unit =
        if newValue > 0 then
            _counter += newValue

    def move(): Unit

    def brake() =
        println("I stop here!")
```

La méthode `move` n'est pas définie dans le trait `Vehicle`. Toute sous-classe de `Vehicle` devra spécifier le comportement concret de `move` dont on ne connaît que le type.

▶ *Faites de `Bicycle` une sous-classe de `Vehicle` et ajoutez une autre sous-classe `Scooter`.*

▶ *Modifiez `travel` pour accepter tout véhicule.*

Une même classe peut hériter d'une seule super-classe, mais de multiples traits.

Par exemple, les déclarations suivantes sont possibles si `B` est une classe et si `C` et `D` sont des traits :

```
class A extends B: ...
class A extends C: ...
class A extends B with C: ...
class A extends B with C with D: ...
```

3 Collections et programmation fonctionnelle

(Book : Domain Modeling → FP Modeling — Collections — Functional Prog.)

Scala propose deux types de structures de données. Le contenu des classes peut être mutable (`scala.collection.mutable._`) ou immuable (`scala.collection.immutable._`).

Nous allons explorer les collections suivantes :

- les tableaux (`scala.Array` équivalent à `scala.collection.mutable.ArraySeq`) ;
- les listes (`scala.collection.immutable.List`) et ;
- les tableaux associatifs (`scala.collection.immutable.HashMap`).

▶ *Avant de lire la suite, explorez les graphes du sous-chapitre « Three main categories of collections » du chapitre « Collections types » du Book.*

▶ *Identifiez dans les graphes où se situent les collections (en fond gris) et à quel trait de haut-niveau elles appartiennent (en fond bleu).*

3.1 Introduction : Les tableaux

(API : `scala.Array`)

Commençons avec un type usuel, les `Array` :

```
val a = Array(1, 3, 5)

println("First element: ${a(1)}")
```

Il existe plusieurs possibilités pour déclarer un tableau :

```
val a = Array(1, 3, 5)
val b = Array[String]("good", "bad", "ugly")
val c = Array.ofDim[Int](3)
```

Les deux premières lignes créent un `Array` avec du contenu. Dans le premier cas, le type est déterminé implicitement par Scala (`Int`). Dans le deuxième cas, on le spécifie explicitement (`String`). Dans le troisième cas, on utilise explicitement le *companion object* afin de créer un tableau de 3 entiers, initialisé avec la valeur 0.

Deux choses à remarquer :

- `Array` (et les autres collections) sont un exemple d'une classe *polymorphe*, paramétrée par un type (tel que `Int`, `String`, ou une classe quelconque car aucune contrainte n'est imposée).
- Dans les exemples ci-dessus, `a`, `b` et `c` sont des *références* à des objets du type `Array[_]`.

  Le fait de déclarer `a` comme `val` veut dire que `a` designera toujours le même objet pendant sa vie, même si le contenu de cet objet peut muter. Ainsi, une déclaration telle que `var f = a` crée simplement une deuxième référence vers l'objet pointé par `a`.

Les tableaux multidimensionnels sont supportés par Scala grâce à la méthode `ofDim`.

Pour créer un `Array` de taille 5 dont les éléments sont des `Array` de `Int` de taille 3 :

```
val g = Array.ofDim[Int](5, 3)

g(4)(2) = 84
```

3.2 Les listes (API : scala.collection.immutable.List)

`List` est un conteneur `immutable`, c'est-à-dire que suite à sa déclaration, toute tentative de modification échoue :

```
val a = List(1, 2, 3, 4)
val b = List(5, 6, 7, 8)

println(3 :: a)
println(a ++ b)

// Erreur:
a(2) = 5
```

`List` est réalisée par des listes chaînées. Du coup, `a(i)` n'est pas une opération en temps constant. De plus, certaines opérations sur les `List` créent de nouveaux objets de type `List`, qui auront un coût de $O(n)$, pour n éléments.

Les différentes collections se distinguent donc par les opérations possibles sur les objets et leur efficacité. Un comparatif des collections permet de sélectionner la collection adéquate.

3.3 Tableaux associatifs (API : scala.collection.immutable.HashMap)

Scala supporte aussi des ensembles associatifs qui stockent des paires (*clé*, *valeur*) ; on peut alors retrouver la valeur à partir de la clé. La classe `HashMap` est alors paramétrée par les types des clés et des valeurs.

La classe `HashMap` n'est pas disponible dans le *prelude* (l'ensemble des classes, fonctions ou variables définies par défaut dans Scala), il va être nécessaire d'importer la classe `HashMap` :

```
import scala.collection.immutable
```

Créons un tableau associatif (par défaut non-mutable) des `String` aux `Int` avec deux paires :

```
val m = immutable.HashMap("a" -> 3, "c" -> 5)

println(m("a"))
```

Créons cette fois-ci un tableau associatif initialement vide mais mutable, en spécifiant les types paramétrés explicitement :

```
import scala.collection.mutable

val n = mutable.Map[String, Int]()

n("a") = 5

println(n)
```

3.4 Opérations

On est souvent amené à traverser des collections de données. Le paradigme fonctionnel permet d'enchaîner les traversement de collections tout en y appliquant des modifications.

Il est possible de récupérer chaque élément d'un tableau grâce à un *for comprehension* :

```
val a = Array(5, 2, 8, 1)

for (i <- a):
    println(i)
```

Une méthode similaire est d'appeler la méthode `foreach` :

```
a.foreach(x => println(x))
a.foreach(println(_))
```

Il est à noter que la seconde forme n'est possible que pour les fonctions anonymes ayant un paramètre qui n'apparaît qu'une seule fois.

Des opérations permettent de générer de nouvelles listes :

```
// Transformation de la liste
a.map(2 * _)

// Filtrer la liste en fonction du prédicat
a.filter(_ >= 5)
```

Des opérations permettent d'effectuer des calculs sur les éléments :

```
a.exists(_ >= 7)
a.find(_ >= 7)
a.count(_ <= 7)
a.foldLeft(0)((a, b) => a + b)
```

▶ *Avant de lire la suite, essayer de trouvez à quoi correspondent ces fonctions grâce à l'`API`.*

▶ *Petit indice : Il faut regarder ici `scala.collection.IterableOnceOps`.*

Quelques remarques :

- Dans la deuxième ligne, le type de retour de `find` est un type `Option[T]`, qui peut soit être `None`, soit `Some(x: T)` où `T` est le type des éléments de `a`. Nous allons détailler cela dans la section 4.3.
- Dans la dernière ligne, 0 est une valeur initiale. La fonction est d'abord appliquée sur la valeur initiale et le premier élement, puis sur le résultat et le deuxième élément et ainsi de suite.

La fonction `reduceLeft` est similaire à `foldLeft`, mais omet la valeur initiale :

```
a.reduceLeft((a, b) => (a + b))
```

On remarque que `a.reduceLeft(_ + _)` fonctionne aussi en plus court, et qu'il existe simplement `a.sum` (ainsi que `a.min` et `a.max`).

▶ *Faites la concaténation des éléments d'un `Array[String]`. Trouvez aussi la longueur de la chaîne la plus courte.*

Dernière remarque, un tableau multidimensionnel peut être ramené à un tableau simple par `flatten`.

3.5 *For comprehensions* (Tour : For comprehension)

Scala offre une notation compacte des boucles imbriqués avec les *for comprehensions*.

Considérons la classe `Cell` qui représente une cellule avec coordonnées x, y contenant une valeur v :

```
class Cell(val x: Int, val y: Int):
    // La valeur de la cellule n'est pas accessible depuis l'extérieur
    private var v: Int = 0

    // On autorise la lecture de v: val x = cell.v
    def v: Int = v

    // On autorise l'écriture de v: cell.v = x
    def v_=(n: Int) =
        v = n
```

La boucle suivante initialise une grille 8×8 de cellules :

```
var g = Array.ofDim[Cell](8, 8)

for (x <- 0 to 7; y <- 0 to 7) do
    g(x)(y) = Cell(x, y)
```

Une *for comprehension* peut aussi filtrer des valeurs :

```
for (x <- 0 to 7; y <- 0 to 7; if x - y == 3) do
    g(x)(y).v = 1
```

Avec le mot-clé `yield`, une *for comprehension* peut construire une liste :

```
val cells = for (x <- 0 to 7; y <- 0 to 7) yield g(x)(y)
```

L'opération précédente peut maintenant être exprimée par l'une des lignes suivantes :

```
for (c <- cells) do
    if (c.x - c.y == 3) then
        c.v = 1

for (c <- cells; if c.x - c.y == 3) do
    c.v = 1

cells.filter(c => c.x - c.y == 3).foreach(_.v = 1)
```

▶ *Mettez `v` à `x + y` dans toutes les cellules de `g`. Calculez la somme de toutes les valeurs `v` dans `g`.*

4 *Objects*, *case class* et *pattern matching*

(Book : Domain Modeling → Tools — Control structures)

4.1 *Companion objects*

(Tour : Singleton objects)

Une méthode n'a pas tout le temps besoin d'une instance d'un objet pour être invoquée.

Par exemple, c'est le cas de la méthode `ofDim` de l'objet `Array` dans la section 3 sur les collections.

Ces méthodes sont dites *statiques*, et sont représentées en Scala à travers les *companion objects* :

```
class Circle(val radius: Double):
    def area: Double = Circle.calculateArea(radius)

object Circle:
    private def calculateArea(radius: Double): Double = Pi * pow(radius, 2.0)

println(Circle(5.0).area)
```

Il est à noter que les parenthèses de la fonction `area` ont été omises. En effet, cette fonction ne prend aucun paramètre et n'effectue aucun effet de bord.

4.2 *Case class*

(Tour : Case classes)

Les *case class* sont des classes avec des propriétés particulières. Il convient de penser leurs instances comme des n-uplets immuables. Regardons l'exemple suivant :

```
case class Complex(re: Double, im: Double):
    def size: Double =
        scala.math.sqrt(re * re + im * im)

    def +(c: Complex) =
        Complex(re + c.re, im + c.im)

val c = Complex(1, 2)
val d = c + Complex(-2, 3)
val e = Complex(1, 2)

println(d)
println(c == e)
```

Un nombre complexe est défini entièrement par ses parties réelle et imaginaire. Ces données sont connues lors de la création d'une instance de `Complex`. Les opérations sur les nombres complexes ne changent pas leur état interne (contrairement à l'exemple de la bicyclette).

Les *case class* sont équipées des opérations suivantes :

- L'opérateur `==` comparant les paramètres de deux instances et non leur identité référentielle ;
- Une méthode `toString` définie automatiquement (voir le résultat de `println`).

Du coup, les paramètres d'une *case class* sont des `val` immuables, et la pratique de déclarer des `var` à l'intérieur est découragée (ceux-ci ne seraient pas pris en compte par `==`).

▶ *Comparer avec le comportement de* `==` *et* `toString` *quand* `Complex` *est une classe ordinaire.*

▶ *Programmer une classe abstraite* `Tree` *avec une méthode* `sum(): Int`*. Déclarer des sous-classes* `Node` *et* `Leaf` *de manière que* `Node(Node(Leaf(2), Leaf(3)), Leaf(5)).sum()` *renvoie* 10.

4.3 *Pattern matching* (Tour : Pattern matching)

Tout comme en OCaml, il est possible d'écrire des *pattern matching*. Le *matching* est particulièrement aisé pour les *case class*. Avec la classe `Tree` de la section 4.2, il est possible d'écrire :

```
val t = Node(Node(Leaf(1), Leaf(2)), Leaf(3))

def sum(t: Tree): Int =
    t match
        case Leaf(v) => v
        case Node(l, r) => sum(l) + sum(r)
```

Lorsque le `match` est le seul élément de la fonction, on peut écrire :

```
def sum: Tree => Int =
    case Leaf(v) => v
    case Node(l, r) => sum(l) + sum(r)
```

Dans chaque `case` d'un *pattern matching*, on peut ajouter des tests sur les valeurs avec la syntaxe :

```
def f: Tree => Int =
    case Leaf(0) => 0
    case Leaf(v) if v < 0 => -1
    case Leaf(v) if v > 0 => 1
    case Node(l, r) => f(l) + f(r)
```

▶ *Définissez des classes représentant une expression arithmétique sur des entiers relatifs avec les opérateurs* `Add`*,* `Mul` *et* `Abs` *(valeur absolue). Définissez une fonction qui évalue l'expression.*

▶ *Ajoutez le cas* `X` *parmi les expressions arithmétiques puis définissez une fonction qui évalue la dérivée d'une expression par rapport à* `X`*.*

5 Programmation polymorphe

(Book : Types and the type system — Contextual abstractions)

Dans la section 3 on a vu des *classes polymorphes* telles que `Array[String]` ou `List[Int]`. Dans ces cas, `Array` et `List` sont des classes qui fournissent une fonctionnalité commune pour des objets de différents types. Dans cette section, on étudie quelques exemples où ces classes paramétrées pourront être utiles, et leur fonctionnement de base.

5.1 Polymorphisme

(Tour : Generic classes — Polymorphic methods)

Disons qu'on a envie de déclarer une classe de graphes. Dans nos graphes chaque sommet possède un nom, et il y a un sommet appelé racine :

```
class Node(val name: String)
class Graph(val root: Node)

val g = Graph(Node("racine"))
println(g.root.name)
```

Jusqu'ici, tout marche bien. Maintenant, supposons qu'on a envie d'utiliser cette classe dans de multiples contextes : par exemple dans un cas, les sommets ne portent que des noms, dans d'autres ils sont équipés d'un poids, dans encore un autre cas il s'agit d'un arbre généalogique où les sommets portent des informations sur des personnes (nom, prénom, date de naissance, ...).

À défaut d'ajouter toutes les informations requises dans tous ces contextes à la seule classe `Node` (ce qui rendrait cette classe lourde et peu lisible), on préfère la création des sous-classes. À priori, aucun souci :

```
class ExtNode(val name: String, val weight: Int) extends Node(name)

val g = Graph(ExtNode("racine", 5))
println(g.root.name)
```

`Graph` accepte bien `n` car `ExtNode` est une sous-classe de `Node`. Mais ne serait-il pas intéressant d'afficher le poids de la racine ? Ajoutons la ligne suivante :

```
println(g.root.weight)
```

Or, cela ne passe pas. En fait, le type inféré pour l'attribut `root` est logiquement `Node` car `root` possède ce type. Or, pour obtenir le poids de `root` il faut que Scala sache qu'il s'agit d'un `ExtNode`.

C'est à ce moment où l'utilité des classes polymorphes s'avère. On change la déclaration ainsi :

```
class Graph[N](val root: N)
```

Ici, `N` est un type qui paramètre la classe `Graph` qui sera spécifié lorsqu'on instancie un graphe. D'ailleurs, `g.root` est de type `N`. Le code suivant marche alors sans problème car le type de `g` devient `Graph[ExtNode]` et `g.root` devient `ExtNode`.

```
val g = Graph(ExtNode("racine", 5))
println(g.root.weight)
```

En effet, dans cet exemple, Scala devine que `N` vaut `ExtNode` dans `Graph(n)` car `n` possède ce type.

5.2 Types bornés (Tour : Upper Type Bounds — Lower Type Bounds)

Supposons que la classe `Graph` devrait fournir des fonctionnalités plus intéressantes, par exemple afficher le nom d'un sommet.

```
class Graph[N](val root: N):
    def printRootName() =
        println(root.name)
```

Or, ceci ne fonctionne plus car `root` est désormais du type `N`, et Scala ne connaît rien sur ce type ; en particulier Scala est incapable d'inférer que ce type possède un `name`.

La solution consiste à spécifier que `N` doit être une sous-classe de `Node`.

```
class Graph[N <: Node](val root: N)
```

Inversement, l'opérateur `>:` spécifie une contrainte de super-classe. Prenons le code suivant :

```
class A

class B extends A:
    def mergeWith[T](x: T) =
        List(x, this)

val a = A()
val b = B()
List(a, b)
b.mergeWith(a)
```

On constate que le type inféré de `List(a, b)` est `List[A]` mais que `mergeWith` donne `List[Any]`. Bien que ces deux listes sont composées des mêmes éléments : lorsque Scala compile la méthode `mergeWith`, l'ancêtre commun plus proche de `T` et `B` n'est que `Any`.

La modification suivante permet de résoudre le problème :

```
class B extends A:
    def mergeWith[T >: B](x: T) =
        List(x, this)
```

On exige alors que `T` soit une super-classe de `B` ce qui permet à Scala d'inférer le type `List[T]` pour `mergeWith`. D'ailleurs, cette contrainte n'empêche pas de passer des sous-classes de `B`, le résultat de `mergeWith` sera toutefois du type `List[B]`.

```
class C extends B

val c = C()
b.mergeWith(c)
c.mergeWith(c)
```

5.3 Bornes multiples (Tour : Upper Type Bounds — Lower Type Bounds)

Revenons sur l'exemple d'un graphe. Désormais on le spécifie avec une liste de sommets et arêtes :

```
class Node(val name: String)
class Edge(val from: Node, val to: Node)
class ExtNode(val name: String, val weight: Int) extends Node(name)

class Graph[N <: Node](nodes: List[N], edges: List[Edge]):
    def outgoing(n: N) =
        edges.filter(_.from == n)

    def successors(n: N) =
        outgoing(n).map(_.to)

val n1 = new ExtNode("n1", 5)
val n2 = new ExtNode("n2", 3)
val edge = new Edge(n1, n2)
val g = new Graph(List(n1, n2), List(edge))

g.successors(n1).foreach(n => println(n.name))
```

Cet exemple fonctionne, mais échoue si on remplace `name` par `weight` dans la dernière ligne ; logiquement car `successors` travaille avec une liste de `Edge` qui ne relie que des `Node`.

Il convient alors de paramétrer la classe `Edge` elle aussi :

```
class Edge[N](val from: N, val to: N)

class Graph[N <: Node](nodes: List[N], edges: List[Edge[N]]):
    ...

g.successors(n1).foreach(n => println(n.weight))
```

Supposons maintenant qu'on veut utiliser la classe `Graph` dans un contexte où les arêtes portent des étiquettes, et qu'on a envie de récupérer les étiquettes sur les arêtes sortantes de `n1`.

On définit alors une sous-classe d'arêtes :

```
class ExtEdge[N](from: N, val label: String, to: N) extends Edge(from, to)

val n1 = ExtNode("n1", 5)
val n2 = ExtNode("n2", 3)
val edge = ExtEdge(n1, "a", n2)
val g = Graph(List(n1, n2), List(edge))

g.outgoing(n1).foreach(e => println(e.label))
```

Mais bien sûr cela échoue car `outgoing` renvoie toujours `List[Edge]`.

La solution consiste alors à donner deux paramètres interdépendants à la classe `Graph` :

```
class Graph[N <: Node, E <: Edge[N]](nodes: List[N], edges: List[E])
```

5.4 Covariance et contravariance (Tour : Variances)

Dans l'exemple précédent, `Edge[A]` et `Edge[B]` sont deux classes distinctes et incomparables. Même si `B` était une sous-classe de `A`, cela ne fait pas de `Edge[B]` une sous-classe de `Edge[A]`, ou inversement.

Parfois, un tel comportement est néanmoins désirable ; regardons l'exemple suivant qui réalise une matrice d'un type `T` quelconque (`f` étant une fonction qui fournit le contenu de la matrice) :

```
class Matrix[T](height: Int, width: Int, f: (Int, Int) => T):
    val data = IndexedSeq.tabulate[T](width,height) { (i, j) => f(i,j) }
```

Disons qu'on utilise une matrice de `A` où `A` possède une donnée `n` entier, et qu'on souhaite calculer les sommes des `n` dans toutes les lignes avec `rowsums` :

```
class A(val n:Int)

def rowsums(m: Matrix[A]) =
    m.data.map(_.map(_.n).sum)

val m1 = Matrix(3, 3, (i, j) => A(i * 3 + j))
println(rowsums(m1).mkString(","))
```

Supposons maintenant qu'on possède une sous-classe `B` de `A`. Puisque toute instance de `B` possède elle aussi un `n`, on veut naturellement utiliser `rowsums` avec une matrice de `B`.

```
class B(n: Int) extends A(n)

val m2 = Matrix(3, 3, (i, j) => B(i * 3 + j))
println(rowsums(m2).mkString(","))
```

Or, cela échoue car `m2` est du type `Matrix[B]` considéré incompatible avec le `Matrix[A]` attendu par `rowsums`. Si on modifie la définition de la matrice ainsi :

```
class Matrix[+T](...):
    ...
```

alors `Matrix[B]` sera considéré comme une sous-classe de `Matrix[A]` (car `B` est sous-classe de `A`), et `rowsums` accepte bien `m2`. Dans ce cas, on dit que `Matrix` est *covariant*.

Dans certains cas rares, on a besoin de l'effet inverse (*contravariance*) : une déclaration de la forme `class Matrix[-T](...)` ferait de `Matrix[A]` une sous-classe de `Matrix[B]` si `B` est sous-classe de `A`.

Vous voilà fin prêt pour développer votre propre projet en Scala !

Rendez-vous la semaine prochaine pour découvrir la librairie graphique, en ayant vérifié que vous avez bien accès à votre compte sur le GitLab du Crans.

En attendant, si vous le souhaitez, continuez de parcourir les différents liens donnés dans ce document.